ROUGH SET LIBRARY
USER'S MANUAL
(ver. 2.0, September '93)

(M.Gawryś, J.Sienkiewicz)

Institute of Computer Science
Warsaw University of Technology
Nowowiejska 15/19, 00-665 Warsaw, Poland

TABLE OF CONTENTS

1. INTRODUCTION

The Rough Set Library (RSL)[1] was created by the authors at the Institute of Computer Science at the Warsaw University of Technology. It was intended as a kernel for any academic or business application using rough set theory concepts. It is a C language routine library. It was written and tested in UNIX environment but can easily be transformed and installed in MS-DOS (see Appendix B). The library was tested as a link-time library on a UNIX workstation and is available in the standard ANSI C source code. The only requirement for an application is that it includes the obligatory header file *rough.h*.

The source code of the library routines is contained in several files. Segmentation into the files corresponds to task segregation and, accordingly, to object segregation. For a detailed description of the source structure see Appendix A. The package contains also:

- the obligatory header file: *rough.h*,
- the header files of all library modules included by *rough.h*.
- some examples of information system data files of the special format,
- some examples of simple applications,
- converter of data files from LERS to format accepted by the RSL: *convert.c*.

The RSL will be soon available by FTP anonymous server. Until then send a request for a free software copy and the PostScript or PCL5 file with the manual to e-mail address: mga@ii.pw.edu.pl. The authors would be pleased to hear any comments and questions.

1.1. ROUGH SET CONCEPTS

Rough set theory has been investigated by many researchers over the last ten years. In some papers as well as in the RSL the starting point for consideration is the concept of information system. An information system can be perceived as a data table describing some objects by means of attribute values. Rows are labeled by objects, columns are labelled by attributes and the table is filled with attribute values. The model of an information system is

[1]The RSL was supported by grant N° 8 0570 91 01 from the Polish State Committee for Scientific Research.

used for representing knowledge and the theory relates to data reduction and analysis. The main problems that rough set theory deals with are: discovering dependencies among data, removing data redundancies and generating decision rules. It turns out that many problems of fundamentally different nature can be reduced to the above-mentioned data table analysis. The theory proved to be significant in many areas of Artificial Intelligence: Machine Learning, pattern recognition and expert systems. It seems promising for the design of switching circuits. It has found many applications in various areas [Słowiński'92].

Rough set theory introduces many notions and corresponding definitions. Since it has been described in detail in many papers and the terminology is well established, this manual will not give detailed information on rough sets. The user should refer to the bibliography [Pawlak'91], [Skowron'92]. Before further reading of this manual the user should make sure he is acquainted with the following concepts:

-information system,
-attribute-value table,
-discernibility matrix,
-approximation of set,
-positive region, dependency of attributes,
-dispensability of attributes,
-core and reduct.

1.2 GENERAL DESCRIPTION

The RSL was meant as a tool for those wanting to build any application using rough set theory model, want to research on the theory itself or just want to analyze their particular data. The library saves their time by providing a virtual rough set machine.

The first task of the library is a maintenance of the data structures for the information system. Since the simple attribute-value table is not the only solution, the library provides routines for keeping data in three different structures: the above mentioned attribute-value table, the discernibility matrix and, proposed by the authors, the reduced discernibility matrix. The user can freely choose between them, use them separately or simultaneously. The information system becomes a C language structure which can be declared, filed, read and stored on disk by the use of the library routines.

The RSL answers all the basic questions that can be asked of the information system using the terminology of rough set theory, among them: approximations of sets, attribute dependencies, various coefficients, cores, etc. Since concepts in this theory are introduced by their mathematical definitions, the optimal algorithms for their computation are not trivial. The RSL solves the problem of algorithms optimization.

There are three crucial tasks of high complexity that are solved by the RSL: reduct finding, rule generation and new object classification. Each is implemented in a separate library module formed by many variations, different strategies and optional versions. Some tool routines help to introduce user-defined strategies.

Computation time and construction of algorithms depends heavily on the information system data structure. Since the library provides three different data structures, it also provides three, often totally different, routines leading to computation of the same answer. It leaves selection to the user. This makes it possible to compare the time and memory effectiveness of competing structures and algorithms. Effectiveness depends heavily on such system parameters as number of objects, number of attributes and size of attribute domains, and even on explicit attribute values.

When we use the information system as a model we usually ask for sets and give sets as parameters. Since a set is not a C language structure RSL provides functions for sets handling and operating.

1.3. IMPLEMENTATION INTENTIONS

During design and coding the library routines the authors developed a certain implementation philosophy. Knowledge of the underlying assumptions will certainly facilitate using the library.

First of all the library was to be flexible, even at the price of an increased number of routines and increased number of parameters. All the selection is left to the user. The authors tried to anticipate all his expectations and nonstandard needs. This main assumption implies some of the following ones.

The library does not supply any INPUT/OUTPUT routines. Any interaction will require design of a user interface.

Since the library was meant, among other uses, as a kernel for various

systems using rough set concepts, speed was one of the most important criteria. Another highly important requirement was clear structure of the source code. Only the lowest level of functions have direct access to the system data. Consequently, the source code is easy to analyze, debug and modify.

1.4 POTENTIAL APPLICATIONS

The RSL can be used like any other C library, leaving the problem of input-output to the user, assuming he is also a C programmer. Any problem that takes rough sets as a model can be successfully implemented.

The RSL may also be of primary importance in testing the relative usefulness of the three different data structures proposed for the information system. The library allows the user to assess implied differences in memory consumption and computation time.

One of the main objectives of the RSL was to provide a kernel for an interactive system, ready be used by an inexperienced computer user. Although there are many possible forms of such application we propose, as an example, to consider only two of them:

A. INTERPRETER OF QUERIES FOR THE INFORMATION SYSTEM.

In this approach programmer has to determine the type of input for the information system. It may be a full-screen editor of the attribute-value table. It can use the standard of file format provided by the RSL. The whole system takes the form of a pull-down menu calling the editor and calling the supplied functions of queries. This is generally very simple to program.

B. EXPERT SYSTEM WITH KNOWLEDGE ACQUISITION MODULE.

The architecture of an Inference Engine and an Explanatory Interface would not be determined by the use of the library. A Data Base should take the form of the information system and a Knowledge Base the form of rules reduced from this system. The library takes the role of a knowledge acquisition module. Routines from the classification module of RSL provide some classification strategies but they can be reinforced with many user-specified ones.

2. DATA STRUCTURES

The RSL defines three types to be used in applications:

setA - set of attributes,
setO - set of objects and
SYSTEM - information system descriptor.

Their definitions are included in the header file *rough.h*. However, their purpose and usage require some explanation.

2.1. SYSTEM

SYSTEM type defines a structure that can be called an information system descriptor. It contains information about the system parameters:
- number of objects,
- number of attributes,
- system name.

The descriptor contains pointers to the information system data matrices:
- MATRIX A,
- MATRIX D,
- MATRIX X,

a pointer to an additional description and some other fields not essential for the user. An application source code should declare pointer to the system descriptor. The library supplies all necessary routines for handling such a descriptor, allocating memory, initializing, reading and writing its fields, storing and retrieving it from the disk, connecting, filling and disconnecting the matrices. There is no need to know the form of system descriptor or the physical structure of its data matrices. It is also possible to design an application in which knowledge of a logical structure of all data matrices in not necessary.

2.2. MATRIX A

MATRIX A is the attribute-value table. This name is sometimes used as a notional equivalent to the information system. However, in the RSL it is only the basic but not the only structure for keeping the information system data. The attribute-value table is a two-dimensional matrix with rows labelled

by objects and columns labelled by attributes. A single element of the matrix is the coded value of the indicated attribute for the indicated object. Missing values are coded by *MINUS*. For the RSL routines missing value matches all other values (is indiscernible with any other value). Since the library does not supply an attribute-value table editor, neither does it specify the way in which the values, usually some symbols, should be coded. Attributes and objects are indicated by their order number of type *int* and values by their code of type *value_type* (predefined as *unsigned int*):

MATRIX_A(3,5)=23

If the application wants to use the attribute-value table in its symbol form:

MATRIX_A(color,5)=green

coding and decoding should be done by the interface. All generated coding tables can be connected to the system descriptor as an additional description (see 2.5). MATRIX A is the only data matrix that is stored with a system descriptor on disk in files of the special format. Two other data matrices can be generated from MATRIX A.

MATRIX A is implemented as an array of *value_type* values (natural sequence) and can be accessed with the use of the RSL routines.

2.3. MATRIX D

MATRIX D is the discernibility matrix, a notion introduced by Skowron [Skowron'92]. It is a square, symmetric matrix with rows and columns labelled by objects. An element of MATRIX D is a set of attributes discerning indicated objects.

example:

MATRIX_D(2,5) = MATRIX_D(5,2) = {1,3,5,6,7,9,13}

MATRIX_D(a,a)=∅

MATRIX D can be generated from MATRIX A and used alternately or simultaneously. Most algorithms works quicker on MATRIX D, but the generation consumes time and requires a lot of memory. Estimating effectiveness of its use is the subject of some research reports and will not be further discussed. Testing such effectiveness may be a goal of some applications. Since MATRIX D is not an equivalent of MATRIX A the library does not supply any routines for storing MATRIX D on disk (the user may do it himself).

MATRIX D is implemented as an array of sets of attributes (*setA*) and can be accessed with the use of the RSL routines (indexing is tricky).

2.4. MATRIX X

MATRIX X is called a reduced discernibility matrix. It was exclusively designed for use in core and reduct computation algorithms. It is an array rather than a matrix, since it has one dimension and its elements are not labelled or sorted. MATRIX X contains elements of MATRIX D (sets of attributes) after removing all oversets (*{a,b}* absorbs *{a,b,c}*) and repetitions. This is not the only reduction achieved by MATRIX X. In the generating routine the user has to specify two parameters *P*,*Q*. They are both sets of attributes. An element of MATRIX D is considered by the overset-absorbing algorithm only if it contains any attributes of *Q* and then as a product with *P*. *P* can be perceived as a set of condition attributes, and *Q* as a set of decision attributes. This is the only way, if MATRIX X is to be used for finding relative core and reducts. The user can also reduce the number of considered objects. MATRIX X is a very specific data structure and can be used only for core and reduct algorithms. MATRIX X is not permanent and can be closed by some rule generation routines. MATRIX X is usually a fraction of size of MATRIX D but the exact size cannot be computed before generation. It is not equivalent to MATRIX D or MATRIX A. For details see the description of the generating routine *InitX()*.

2.5. ADDITIONAL DESCRIPTION

So-called additional description is a field of system descriptor that can indicate all the information system data structures that are not part of the RSL standard. The library supplies routines for connecting such structure. Routines storing the information system to disk handle the additional description as well. The RSL does not supply any direct tool for an information system editing. An attribute-value table is accepted in integer-coded form. User-implemented editing, and especially transforming an attribute-value table from its symbolic form to a coded form used by MATRIX A, produces some extra data. Such data, for example symbolic names of attributes and domains, are not essential for the RSL but are necessary for the user interface and can be

connected to system descriptor as the additional description.

2.6 SETS

It is difficult to imagine an application that would not use sets. Sets are parameters and results of most routines. There are two domains of sets in the information system: attributes and objects. Both attributes and objects are indicated by their index in MATRIX A, but their total number is different. Type *setA* is therefore used for sets of attributes and *setO* for sets of objects. Both are implemented as bitmaps on integral type *cluster_type* (predefined as *unsigned int)*.

2.7 OUTPUT

Some routines of the RSL produce large data structures. The most common are: reducts and rules.

Reduct is simply a set of attributes (*setA*). The user can access a collection of reducts produced by a routine as an array of sets. An array index should be incremented by the size of the set of attributes (*SizeSetA() * sizeof(cluster_type)*). The easiest way is to use relevant macros (see section 4.2).

Single rule looks like a single raw from the attribute-value table so the collection of rules produced by any routine has a structure of MATRIX A. Reduced values of attributes are coded by predefined value *MINUS*. The easiest excess to collection of rules is supported by macro *ElemOfRule()*.

3. OTHER TOPICS

In this chapter the user can find :

- a complete collection of predefined types and values,
- a description of information system data file format,
- error handling strategy description.

3.1 PREDEFINED TYPES

Data types for use in application are described in the previous chapter:

SYSTEM, setO ,setA.

The user has access to other two types crucial for implementation.

value_type

stores a single value of attribute. It is predefined as *unsigned int*. It is used for attribute-value table and rule implementation. Predefined value *MINUS* is reserved for missing or reduced elements.

cluster_type

is an integral type for bitmap implementation. It is predefined as *unsigned int*. It is used for set, reduct and discernibility matrix implementation. Its size should provide the optimal bitwise operation speed.

Both *value_type* and *cluster_type* can be redefined in the file *rough.h* which causes the need for the library recompilation. Both types must stay integral. Their size affects the size of data structures and speed of most routines.

3.2 PREDEFINED VALUES AND GLOBAL VARIABLES

Most of the query routines take the source data matrix as a parameter of type *int.* Predefined values are:

MATA, MATD, MATX.

One of *value_type* values is reserved for missing elements:

MINUS

The library routine *SelectRules*() has an option of type *int* with predefined values:

FASTOPT, BESTOPT.

All system global variable identifiers start with underscore. The user should

avoid such names.

3.3 DATA FILE FORMAT

The information system in the form of MATRIX A can be stored on disk in text file. If the user intends to save and retrieve such files only with the use of the library routines the format is not important.

Such files can be produced with any common text editor. If the user plans the design of the specialized information system data editor we propose to consider the RSL format as a base disk format. The main difference between our format and many others is that values has to be coded as integers. Absence of a single value is coded by character *?*.

Information System Special File Format:

NAME: <string> # max 50 characters
ATTRIBUTES: <integer> # n - number of attributes
OBJECTS: <integer> # m - number of objects

$<a_{0,0}>$ $<a_{0,1}>$ $<a_{0,2}>$ ... $<a_{0,n-1}>$
$<a_{1,0}>$ $<a_{1,1}>$ $<a_{1,2}>$ ... $<a_{1,n-1}>$
...
$<a_{m-1,0}>$ $<a_{m-1,1}>$ $<a_{m-1,2}>$ ... $<a_{m-1,n-1}>$

<text> # additional description to the end of file

where $a_{x,y}$ = value of attribute y for object x, integer or ? for missing value

An additional description contents are not part of the standard but it may store a coding table of attribute values and names or nay other user-defined structures. For an example of the system file see Appendix C.

3.4 ERROR HANDLING

The problem of error-handling has been left to particular implementations, and consequently the library does not have an internal error-handling sub-system. The user should design a specific error-handling strategy and the library only provides some tools for joining it with our routines. The library routines detect only some errors (for example wrong parameters, uninitiated data matrices, some memory allocation errors). Most of the RSL routines return *int* value. In case of an error, they return the negative code (-code) of the error. All routines, in case of an error , set a global *int* variable *_rerror* (definition in *rough.h*) to an error code (positive). Variable *_rerror* can be used in an application error handling strategy. It is set to zero only once, at the beginning of a program. The error messages and an example of a macro that outputs them are contained in the file *rerror.h*. It can be optionally included in the application or just viewed during debugging. It is not used by any library routine.

Error codes:

```
char *errcodes[10]={"everything O.K.",
                    "Cannot open file",
                    "Wrong format of file",
                    "Not enough memory",
                    "Cannot write to file",
                    "Matrix A not initialized",
                    "Matrix D not initialized",
                    "Matrix X not initialized",
                    "Wrong matrix type",
                    "Set element out of domain"
                      };
```

4. LIBRARY ROUTINES BY CATEGORY

The library routines are grouped into eight categories:

- system control routines,
- data access routines,
- set handling routines,
- basic queries,
- core finding and reduct checking,
- reduct finding,
- rule generation,
- classification.

Each category has a specific method of placing parameters, returning result etc. There is some information about each category that is necessary for the proper use of routines.

4.1. SYSTEM CONTROL ROUTINES

Routines from this category enable a wide range of operations on the information system. A first parameter is always a pointer to the system descriptor. The user can keep many initialized system descriptors, that is, many information systems, but only one is active at any given moment. The system is activated by function *UseSys(SYSTEM *sys)*. From this moment on, all functions from other categories will work on the parameters and data of the activated information system. For example, sets should be initialized, handled and closed in the range of the same active system.

Asize	**FileToSys**	**MatMemSize**
AttributesNum	**FillAfromAscii**	**ObjectsNum**
CloseMat	**InitD**	**PutToA**
CloseSys	**InitEmptySys**	**SetName**
ConnectA	**InitX**	**SetParameters**
ConnectDescr	**InitXforObject**	**SysName**
Description	**InitXforObjects**	**SysToFile**
DisconMat	**InitXforObjectFromClass**	**UseSys**
Dsize	**MatExist**	**Xsize**

Example 1

```
SYSTEM *sys1;  /* pointer to system descriptor */
...
sys1=InitEmptySys(); /* creating system descriptor */
FileToSys(sys1,"HEP.SYS"); /* importing a system */
InitD(sys1);   /* generating MATRIX D */
UseSys(sys1);  /* assigning the active system */
...
/* following queries will work on an information  */
/* system sys1 imported from file HEP.SYS and on  */
/* both its MATRIX A and MATRIX D            */
```

Example 2

```
SYSTEM *sys2;
...
sys2=InitEmptySys();
SetParameters(sys2,...);
               /* assigning system parameters */
SetName(sys2,...); /* assigning system name */
ConnectA(sys2,malloc(Asize(sys)));
               /* allocating and connecting */
               /* an empty MATRIX A */
for(attribute...)
  for(object...)
     { value=...;
       PutA(sys2,object,attribute,value);
     }         /* filling MATRIX A */
UseSys(sys2);
/* active system sys2 was created in application  */
/* and has only MATRIX A                     */
```

4.2. DATA ACCESS ROUTINES

If a collection of queries supplied by the library is not satisfactory or an application needs a direct access to the data matrices for any other reason, such a facility is provided. A stream access to MATRIX X and MATRIX D or a collection of reducts is organized with the use of some macros (names with capital letters). All routines access the active information system.

CompareA	**GetA**	**SingCompA**
CompareD	**GetD**	**SingCompD**
ElemOfRule	**GetDfromA**	**START_OF_MAT**

START_OF_D	**ELEM_OF_MAT**	**END_OF_MAT**
START_OF_X	**NEXT_OF_MAT**	

4.3. SET-HANDLING ROUTINES

All set-handling routines are duplicated for sets of objects and sets of attributes. Functions with the *setO* suffix work on sets of objects (type *setO*). Functions with the *setA* suffix work on sets of attributes (type *setA*). Attributes and objects are indicated by their index in MATRIX A (type *int* starting with 0). Types *setA* and *setO* are implemented as pointers and should therefore be initialized and, at the end, freed, using the library routines. Since information system parameters are important for sets implementation, sets should be initialized and handled after the *UseSys()* function, in the range of a single active system. All functions that result in set valuation should be given this set, initialized, as a first argument.

AddSetA	**ContSetA**	**InterSetA**
AddSetO	**ContSetO**	**InterSetO**
AndSetA	**CopySetA**	**IsEmptySetA**
AndSetO	**CopySetO**	**IsEmptySetO**
ArgToSetA	**DelSetA**	**NotSetA**
ArgToSetO	**DelSetO**	**NotSetO**
AttrValSetO	**DifSetA**	**OrSetA**
CardSetA	**DifSetO**	**OrSetO**
CardSetO	**FillSetA**	**PrintSetA**
ClassSetO	**FillSetO**	**PrintSetO**
ClearSetA	**InitEmptySetA**	**SizeSetA**
ClearSetO	**InitEmptySetO**	**SizeSetO**
CloseSetA	**InitFullSetA**	**TabToSetA**
CloseSetO	**InitFullSetO**	**TabToSetO**
CompSetA	**InSetA**	
CompSetO	**InSetO**	

Example 1

```
setA P,Q;
setO result;
...
UseSys(...); /* activating the system */
...
result=InitEmptySetO(); /* initializing empty sets */
P=InitEmptySetA();
Q=InitEmptySetA();
...
ArgToSetA(P,3,0,1,2);     /* P={0,1,2}      */
ArgToSetA(Q,4,3,4,5,6);  /* Q={3,4,5,6}    */
Pos(result,P,Q);          /* result=POSP(Q) */
PrintSetO(result);        /* set to screen */
```

Example 2

```
setA P;
SYSTEM *sys
...
/* If one do not like the format of PrintSetA() */
/* he can output a set himself */
...
for(a=0; a<AttributesNum(sys); a++);
  if( ContSetA(P,a) )
     { /* output a */ }
```

4.4. BASIC QUERIES

Functions from this category answer the wide range of queries one can ask the active information system using notions from rough set theory. The same queries can be computed from different information system data structures. Matrix type is always the last parameter and its predefined values are:

- MATA O
- MATD 1
- MATX 2

X version is unavailable for all basic queries which means that MATRIX X is not sufficient for their computation. All queries access the active information system. If a routine is to find a set as a result, this set should be initialized and placed as a first parameter. Old contents are not important and will be lost.

AccurCoef	**DependCoef**	**SignifCoef**
Bound	**LowAppr**	**SignifRelCoef**
CardCoef	**Pos**	**UppAprr**
ClassCoef		

4.5 CORE FINDING AND REDUCT CHECKING

Functions from this module find the core of the active information system and check if a given set of attributes is orthogonal, discriminate all objects or is a reduct. MATRIX X version of *core* function gives the same result as other versions only if MATRIX X was generated from the full set of attributes relatively to the full set of attributes (generation parameters P,Q should contain all attributes). Otherwise it will find a relative core.

Core	**IsCoverRel**	**IsRed**
CoreRel	**IsOrtho**	**IsRedRel**
IsCover	**IsOrthoRel**	

4.6 REDUCT FINDING ROUTINES

Functions from this category find and select reducts. The number of reducts of the information system can vary from single one to hundreds of thousands. Computation time and memory consumption may be critical in many application. Consequently the RSL provides many varieties of reduct finding algorithm: finding only the shortest ones, only single ones, ones containing a given set of attributes and many others. Some of them do not guarantee success in finding reducts, but guarantee to find nearly-reducts (we call them pseudo-reducts) in a limited time and in limited numbers. Others implement heuristic search.

The reduct finding function should be given the pointer to the uninitialized set of attributes as a first parameter. The routine will allocate necessary size of memory for the collection of reducts and return their number as a result. The user can access the array of computed reducts with some macros (see 4.2) and shall not forget to free memory.

X version gives the same result as other versions only if MATRIX X was generated from the full set of attributes relatively to the full set of attributes (generation parameters P,Q should contain all attributes). All queries

concern the active information system.

Red	**RedLess**	**RedSingle**
RedRel	**RedRelLess**	**RedRelSingle**
RedFirst	**RedOptim**	**SelectsAllShort**
RedRelFirst	**RedRelOptim**	**SelectOneShort**
RedFew	**RedSetA**	
RedRelFew	**RedRelSetA**	

Example

```
setA red;
int n;
...
UseSys(sys1);
...
n=Red(&red,MATA);
printf("number of reducts : %i\n",n);
for(START_OF_MAT(red,n);END_OF_MAT;NEXT_OF_MAT)
  PrintSetA(ELEM_OF_MAT);
```

4.7 RULE GENERATION

Rule generation in the RSL is based on the attribute value reduction. Lets look at the single object. It is described by many attributes. Some of them are dispensable for classification. If we are only interested in classifying the object to one of the classes discriminated by the set of decision attributes Q (usually single attribute) many attribute values of many objects can be reduced. The set of attributes indispensable for the object to discriminate it from all other objects that do not belong to its class is called a reduct of values. There can be many possible reducts of values for a single object. So from a single object we can derive many rules. If rules obtained from two objects are equal one can be reduced. Consequently the number of rules can be many times smaller or many times greater than the number of objects. The RSL provides many rule generation strategies, differing in expected number of rules, time of computation and level of optimization. The prediction of new objects is not the only possible application of rules and only some strategies will work good for prediction.

All rule generating routines allocates memory for the collection of rules and return their number as a result. The structure of rules is similar to structure

of MATRIX A. Reduced values are coded by predefined value MINUS.

AddRule	**BestRulesForClass**	**Rules**
AllRules	**FastRules**	**RulesForClass**
AllRulesForReducts	**RuleCopy**	**SelectRules**
BestRules	**RuleEQ**	**VeryFastRules**

Example

```
int n, attr;
value_type *rules;
...
n=Rules(&rules,...);
printf("the last rule:\n");
for(attr=0;attr<AttributesNum(sys);attr++)
  if(ElemOfRule(rules,n-1,attr)==MINUS)
    printf(" *  ");
  else
   printf("%4i",ElemOfRule(rules,n-1,attr);
printf("\n");
```

4.8 CLASSIFICATION

Rules generated by the RSL can be used to predict values of decision attributes for new objects (classification). If only one rule matches the new object the algorithm is quite simple. It can happen that none of the rules or more than one matches. The RSL provides three strategies for rule conflict resolution. The user can implement his own strategies using some tool routines from this module.

Classify1	**CompareToRule**	**ObjectsForRule**
Classify2	**CompareToRules**	**StrengthOfRule**
Classify3	**DecisionEQ**	**StrengthOfRules**

Example

```
setA P,Q;
value_type rules;
int n,index;
...
n=Rules(&rules,P,Q,MATD);
index=Classify1(GetA(sys,0,0),rules,n,P,Q);
```

```
if(DecisionEQ(GetA(sys,0,0),ElemOfRule(rules,index,0),Q)
       printf("first object correctly classified/n");
     else
       printf("incorrect classification for first
object/n");
```

5. DESCRIPTION OF LIBRARY ROUTINES

AccurCoef

function	Compute an accuracy coefficient.
syntax	float AccurCoef(setO *X*,setA *P*,int *matrix_type*);
remarks	Returns accuracy of *X* with respect to *P* in the active information system.
def.	$\alpha_P(X) = card(\underline{P}(X)) \,/\, card(\overline{P}(X))$
domain	*matrix_type*: MATA, MATD.

AddRule

function	Adds a new rule to an array.
syntax	void AddRule(value_type **rules*, int **size*, value_type *rule*);
remarks	Adds new *rule* to the collection of *rules*. If rule is unique *size* is incremented.

AddSet

function	Adds a single element to a set.
syntax	int AddSetA(setA *set*, int *attr*); int AddSetO(setO *set*, int *obj*);
remarks	*set* must be initialized. Returns negative error code if element out of domain. Domain determined by the active information system parameters.

AllRules

function	Finds all possible rules.
syntax	int AllRules(value_type ***rules*, setA *P*, setA *Q*, int *matrix_type*);
remarks	For each object from the active information system finds all *Q*-relative reducts of values on *P*. Removes repetitions. Allocates memory and returns number of generated rules.
domain	*matrix_type*: MATA, MATD.

AllRulesForReducts

function	Finds all possible rules for each reduct separately.
syntax	int AllRulesForReducts(value_type ***rules*, cluster_type **reducts*, int *N*, setA *Q*, int *matrix_type*);
remarks	For each reduct form the collection *reducts* of *N* reducts, for each object from the active information system finds all *Q*-relative reducts

of values. Removes repetitions. Allocates memory and returns number of generated rules.

domain *matrix_type*: MATA, MATD.

AndSet

function Finds a product of two sets.

syntax void AndSetA(setA *and*, setA *s1*, setA *s2*);
void AndSetO(setO *and*, setO *s1*, setO *s2*);

remarks Product of *s1* and *s2* is placed to *and*. All parameters should be initialized in the range of the same currently active information system. *and* can be the same set as *s1* or *s2*.

ApprRules

function Finds a collection of rules for approximated classes.

syntax int ApprRules(value_type **rules*, setA *P*, setA *Q*, int *option*, int *matrix_type*);

remarks For each approximated class of objects, for each object find the *Q*-relative reduct of values on *P*. Removes repetitions. Allocates memory and returns the number of generated rules.

domain *matrix_type*: MATA, MATD.

options LOWER - lower approximation (collection of certain rules)
UPPER - upper approximation (collection of possible rules)
NORMAL - no approximation (collection of standard rules)

ApprRulesForClass

function Finds a collection of rules for approximated class.

syntax int ApprRules(value_type **rules*, setO *class*, setA *P*, setA *Q*, int *option*, int *matrix_type*);

remarks For each object from approximated *class* finds a *Q*-relative reduct of values on *P*. Allocates memory and returns the number of generated rules.

domain *matrix_type*: MATA, MATD.

options LOWER - lower approximation (collection of certain rules)
UPPER - upper approximation (collection of possible rules)
NORMAL - no approximation (collection of standard rules)

ArgToSet

function Puts variable number of elements to a set.

syntax void ArgToSetA(setA *set*, int *num*, ...);

void ArgToSetO(setO *set*, int *num*, ...);

remarks — Old contents of *set* will be erased. *num* parameters of function of type int will be put to *set*. Domain of *set* determined by the active information system.

Asize

function — Number of elements in MATRIX A.

syntax — int Asize(SYSTEM **sys*);

remarks — MATRIX A does not have to be connected to *sys*. Size calculated from *sys* parameters. Single element type *value_type*.

AttributesNum

function — Returns a number of attributes in a system.

syntax — int AttributesNum(SYSTEM **sys*);

remarks — The number of attributes is the information system parameter. It can be set by SetParameters() or FileToSys() routines.

AttrValSet

function — Finds a set of objects that have a given value of a given attribute in MATRIX A.

syntax — void AttrValSetO(setO *set*, int *attr*, value_type *val*);

remarks — The active information system should have MATRIX A connected. All objects that have value *val* on attribute *attr* will be placed to *set*. *set* must be initialized but its original contents will be lost.

BestRules

function — Finds minimal set of rules.

syntax — int BestRules(value_type ***rules*, setA *P*, setA *Q*, int *matrix_type*);

remarks — For each object from the active information system finds all the shortest *Q*-relative reducts of values on *P*. Then selects the minimal collection of rules that cover all objects. Allocates memory and returns number of generated rules.

domain — *matrix_type*: MATA, MATD.

BestRulesForClass

function — Finds minimal set of rules.

syntax — int BestRulesForClass(value_type ***rules*, setO *class*, setA *P*, setA *Q*, int *matrix_type*);

remarks — For each object from the *class* finds all the shortest *Q*-relative reducts of values on *P*. Then selects the minimal collection of rules that cover all objects. Allocates memory and returns number of generated rules.
domain — *matrix_type*: MATA, MATD.

Bound

function — Finds a boundary.
syntax — int Bound(setO *bound*, setO *X*, setA *P*, int *matrix_type*);
remarks — Puts *P*-boundary of *X* from the active information system into the *bound*. All sets should be initialized in the range of this system.
def. — $BN_P(X)= \overline{P}(X) - \underline{P}(X)$
domain — *matrix_type*: MATA, MATD.

CardSet

function — Counts elements of a set.
syntax — int CardSetA(setA *set*);
int CardSetO(setO *set*);
remarks — Returns cardinality of *set*.

ClassCoef

function — Computes quality of classification.
syntax — float ClassCoef(setO *X*, setA *P*, int *matrix_type*);
remarks — Returns the quality of classification {*X*,-*X*} with respect to *P* in the active information system. All sets should be initialized in the range of this system.
def. — $\gamma_P(X)= card(\underline{P}(X) \cup \underline{P}(-X)) / card(all\ objects)$
domain — *matrix_type*: MATA, MATD.

Classify1

function — Chooses the best rule to cover sample. Strategy no. 1.
syntax — int Classify1(value_type **sample*, value_type **rules*, int *N*, setA *P*, setA *Q*);
remarks — Finds the most frequent decision *Q* among *rules* with minimal distance from a given *sample*. Distance defined as a number of unmatched attributes from *P*. *N* - number of rules. Returns index of the rule with the chosen decision.

Classify2

function — Chooses the best rule to cover sample. Strategy no. 2.

syntax int Classify2(value_type **sample*, value_type **rules*, int *N*,
setA *P*, setA *Q*);

remarks Selects all rules with minimal distance from a given *sample*. Chooses the decision which appears in the largest number of objects matched by any of selected rules. Distance defined as a number of unmatched attributes from *P*. *N* - number of rules. Returns index of the rule with the chosen decision.

Classify3

function Chooses the best rule to cover sample. Strategy no. 3.

syntax int Classify3(value_type **sample*, value_type **rules*, int *N*,
setA *P*, setA *Q*);

remarks Selects all rules with minimal distance from a given *sample*. For every selected rule sums the number of matched objects. Chooses the decision with the largest total sum. Distance defined as a number of unmatched attributes from *P*. *N* - number of rules. Returns index of the rule with the chosen decision.

ClassSet

function Finds a class of objects.

syntax int ClassSetO(setO *class*, int *obj*, setA *Q*)

remarks Finds a set of objects from active information system indiscernible with *obj* on *Q*.

ClearSet

function Clears a set.

syntax void ClearSetA(setA *set*);
void ClearSetO(setO *set*);

remarks *set* will be emptied.

CloseMat

function Closes a matrix.

syntax void CloseMat(SYSTEM **sys*, int *matrix_type*);

remarks Disconnects the matrix from *sys* and frees memory.

domain *matrix_type*: MATA, MATD, MATX.

CloseSet

function Frees memory allocated for a set.

syntax void CloseSetA(setA *set*);

void CloseSetO(setO *set*);

<u>remarks</u> All sets that have been initialized must be freed before program end or before initialization for a new active information system.

CloseSys

<u>function</u> Closes a system.

<u>syntax</u> void CloseSys(SYSTEM **sys*);

<u>remarks</u> Frees memory for all matrices that are connected to *sys* and then frees memory used by system descriptor.

CompareA

<u>function</u> Compares two objects in MATRIX A on a set of attributes.

<u>syntax</u> int CompareA(int *obj1*, int *obj2*, setA *P*);

<u>remarks</u> Returns 1 if objects *obj1* and *obj2* are equal on all attributes from *P*. Otherwise returns 0. MATRIX A should be connected to the active system.

CompareD

<u>function</u> Tests discernibility of two objects in MATRIX D.

<u>syntax</u> int CompareD(int *obj1*, int *obj2*, setA *P*);

<u>remarks</u> Returns 1 if objects *obj1* and *obj2* are equal on all attributes from *P*. Otherwise returns 0. MATRIX D should be connected to the active system.

CompareToRule

<u>function</u> Compares a rule to a sample.

<u>syntax</u> int CompareToRule(value_type **sample*, value_type **rule*, setA *P*);

<u>remarks</u> Compares *rule* to a *sample* on attributes *P*. Returns the number of unmatched attributes. Reduced or missing values (MINUS) matches all.

CompareToRules

<u>function</u> Compares a sample to a collection of rules.

<u>syntax</u> int *CompareToRules(value_type **sample*, value_type **rule*,int *N*, setA *P*);

<u>remarks</u> Compares a collection of *N rules* to a *sample* on attributes *P*. Creates the table of distances (unmatched attributes). Allocates memory (size = *N**sizeof(int). Reduced or missing values (MINUS) match all.

CompSet

function	Compares two sets.
syntax	int CompSetA(setA *s1*,setA *s2*); int CompSetO(setO *s1*,setO *s2*);
remarks	Returns 1 if *s1* and *s2* are identical, otherwise returns 0.

ConnectA

function	Connects MATRIX A to a system descriptor.
syntax	void ConnectA(SYSTEM **sys*, value_type **buf*);
remarks	The buffer *buf* (MATRIX A) does not have to be already filled with proper values yet, but its size should correspond to system parameters.
example	ConnectA(sys1,malloc(MatMemSize(sys1,MATA)));

ConnectDesc

function	Connects additional description to system descriptor.
syntax	void ConnectDescr(SYSTEM **sys*, void **descr*, int *size*);
remarks	The additional description is just a block of memory.

ContSet

function	Tests if a set contains an element.
syntax	int ContSetA(setA *set*, int *attr*); int ContSetO(setO *set*, int *obj*);
remarks	Returns 1 if element (*attr,obj*) is in *set*, otherwise returns 0.

CopySet

function	Copies a set.
syntax	void CopySetA(setA *dest*, setA *source*); void CopySetO(setO *dest*, setO *source*);
remarks	Copies *source* to *dest*. Both sets should be initialized in the range of the currently active information system.

Core

function	Finds a core.
syntax	int Core(setA *core*, int *matrix_type*);
remarks	Puts a core of the active information system to *core*. *core* should be initialized in the range of the same system. X version gives the same result only if MATRIX X was generated with parameters P and Q containing all attributes. Otherwise X version finds Q-relative core of

	the information system with attributes restricted to P. If error detected, returns negative error code.
def.	*CORE= {a∈R: a is indispensable in R}* where *R= {all attributes}*
domain	*matrix_type*: MATA, MATD, MATX
example	InitX(...,P,Q,..); ... CoreX(core); /* == CoreRelA(core,P,Q) */

CoreRel

function	Finds a relative core.
syntax	int CoreRel(setA *core*, setA *P*, setA *Q*, int *matrix_type*);
remarks	Puts to *core* a *Q*-relative core of the active information system with attributes restricted to *P*. This reflects situation when *P* represent condition attributes and *Q* decision attributes. *Q* does not have to be a complement of *P*. All attributes should be initialized in the range of the same system. If error detected, returns negative error code. For X version see Core.
def.	$CORE_Q(P)= \{a \in P\text{: a is Q-indispensable in P}\}$
domain	*matrix_type*: MATA, MATD

DecisionEQ

function	Compares two rules.
syntax	int DecisionEQ(value_type *rule1*, value_type *rule2*, setA *Q*);
remarks	Returns 1 if *rule1* and *rule2* match on *Q*. Otherwise returns 0.

DelSet

function	Deletes an element from a set.
syntax	int DelSetA(setA *set*, int *attr*); int DelSetO(setO *set*, int *obj*);
remarks	No effect if element (*attr,obj*) is not in *set*. Returns error code if element out of domain. Domain determined by the active information system.

DependCoef

function	Computes a degree of dependency.
syntax	float DependCoef(setA *P*, setA *Q*, int *matrix_type*);
remarks	Returns degree of dependency *Q* from *P* in the active information system. All sets should be initialized in the range of the same system.

def. $\gamma_P(Q) = card(POS_P(Q)) / card(\{all\ objects\})$
domain *matrix_type*: MATA, MATD.

Description
function Returns an additional description of a system.
syntax void *Description(SYSTEM **sys*);
remarks Returns a pointer to an additional description connected to *sys*. This is a block of memory with no predefined structure.

DifSet
function Finds a complement of a set.
syntax void DifSetA(setA *dif*, setA *s1*, setA *s2*);
void DifSetO(setO *dif*, setA *s1*, setA *s2*);
remarks Puts a complement of *s2* in *s1* to *dif*. All sets should be initialized in the range of the currently active information system. *dif* can be the same set as *s1* or *s2*.

DisconMat
function Disconnects matrix from the system descriptor.
syntax void DisconMat(SYSTEM **sys*, int *matrix_type*);
remarks Does not free memory used by matrix. If necessary, it should be freed before disconnecting or closed by CloseMat().

Dsize
function Number of elements in MATRIX D.
syntax int Dsize(SYSTEM **sys*);
remarks MATRIX D does not have to be connected to *sys*. Size calculated from *sys* parameters. Single element type: *setA*.

ELEM_OF_MAT (macro)
function One of macros enabling stream access to an array of sets of attributes.
see START_OF_D,START_OF_X,START_OF_MAT

ElemOfRule (macro)
function Value of an attribute from a rule.
syntax value_type ElemOfRule(value_type **rules*, int *num*, int *attr*);
remarks The macro indexes a collection of rules produced by the RSL. The rule number *num* (the first rule index 0) on attribute *attr*. Reduced values coded by MINUS.

example

```
value_type *rules;
int n, attr;
...
n=Rules(&rules,...);
printf("the last rule:\n");
for(attr=0;attr<AttributesNum(sys);attr++)
  if(ElemOfRule(rules,n-1,attr)==MINUS)
    printf("*   ");
else
    printf("%4i",ElemOfRule(rules,n-1,attr);
printf("\n");
```

END_OF_MAT (macro)

function One of macros enabling stream access to an array of sets of attributes.

see START_OF_D,START_OF_X,START_OF_MAT

FastRules

function Quickly finds a collection of rules.

syntax int FastRules(value_type ***rules*, setA *P*, setA *Q*, int *matrix_type*);

remarks For each object from the active information system finds the shortest *Q*-relative reduct of values on *P*. Removes repetitions. Allocates memory and returns number of generated rules.

domain *matrix_type*: MATA, MATD.

FileToSys

function Imports a system from a file of the special format.

syntax int FileToSys(SYSTEM **sys*, char **filename*);

remarks For the precise format standard see 2.7. File can be created in application or with the help of SysToFile() routine. It contains information system parameters, a name, an additional description and MATRIX A. A system descriptor should be initialized and empty. If error detected returns negative error code.

FillAfromAscii

function Fills MATRIX A with values from an ASCII file.

syntax int FillAfromAscii(SYSTEM **sys*, FILE **file*);

remarks Fills MATRIX A that is connected to *sys*. Number of attributes (columns) and number of objects (rows) determined by *sys* parameters. The routine will read from the current position of the *file*

decimal numbers from domain *value_type* separated by white characters (space, end of line). Missing values coded by -1. If error detected returns negative error code.

FillSet

<u>function</u> Puts all elements of domain to a set.

<u>syntax</u> void FillSetA(setA *set*);
void FillSetO(setO *set*);

<u>remarks</u> *set* should be initialized. It will contain all elements of domain of the active information system.

GetA

<u>function</u> Gets a single element of MATRIX A.

<u>syntax</u> value_type GetA(int *obj*, int *attr*);

<u>remarks</u> Returns a value of the attribute *attr* of the object *obj* from MATRIX A of the active information system.

GetD

<u>function</u> Gets a single element of MATRIX D.

<u>syntax</u> setA GetD(int *obj1*, int *obj2*);

<u>remarks</u> Returns a set of attributes which is the element of MATRIX D for objects *obj1* and *obj2*. This set should not be altered. MATRIX D should be connected to the active information system.

GetDfromA

<u>function</u> Generates a single element of MATRIX D.

<u>syntax</u> int GetDfromA(setA *elem*, int *obj1*, int *obj2*);

<u>remarks</u> Generates a set of attributes which is the element of MATRIX D for objects *obj1* and *obj2*. This set should be given initialized. MATRIX A should be connected to the active information system. Returns negative error code on error.

InitD

<u>function</u> Generates MATRIX D from MATRIX A.

<u>syntax</u> int InitD(SYSTEM **sys*);

<u>remarks</u> MATRIX A must exist. Memory will be allocated, connected to *sys* and filled. If any error, returns a negative error code.

InitEmptySet

function	Initialize an empty set.
syntax	setA InitEmptySetA(void); setO InitEmptySetO(void);
remarks	Allocates memory for set representation. Memory size determined by active information system parameters. All further operation on this set only in a range of the same system. For use in the range of another system, this set should be closed and initialized once again.

InitEmptySys

function	Initialize a system descriptor.
syntax	SYSTEM *InitEmptySys(void);
remarks	Allocates memory for a system descriptor, initialize its fields as empty and returns a pointer.

InitFullSet

function	Initialize a full set.
syntax	SetA InitEmptySetA(void); SetO InitEmptySetO(void);
remarks	Allocates memory for a set and initialize it with all elements of domain. Memory size and domain determined by the active information system. All further operations on this set, only in a range of the same system. For use in the range of another system, this set should be closed and initialized once again.

InitX

function	Generates MATRIX X from another matrix.
syntax	int(SYSTEM **sys*,setA *P*,setA *Q*,int *matrix_type*);
remarks	MATRIX X was designed exclusively for use in core and reducts queries. It contains all information from information system, that is necessary for computing cores and reducts from the set of attributes *P*, relatively to the set of attributes *Q*. In other words *P* contains condition attributes and *Q* decision attributes. If one wish to compute nonrelative cores and reducts of the whole information system MATRIX X should be initialized with *P* and *Q* containing all attributes. In order to reduce a number of attributes in information system to *P*, but compute nonrelative cores and reducts, MATRIX X should be initialized with Q containing all attributes. MATRIX X is not equivalent to MATRIX D since it is determined by *P* and *Q*.

Computing relative core and reducts from MATRIX A or MATRIX D gives the same result as computing nonrelative core and reducts from MATRIX X. Memory for MATRIX X will be allocated automatically and it will be connected to *sys*. (Algorithm performs oversets and repetitions absorbing on potential elements of MATRIX D. An elements is considered by the algorithm only if it contains any element of *Q* and then as a product with *P*).

<u>domains</u> *matrix_type*: MATA, MATD

<u>example</u>
```
InitX(sys,P,Q,MATD);
UseSys(sys);
......
CoreX(core)  /* == CoreRelD(core,P,Q) */
```

InitXforObject

<u>function</u> Generates MATRIX X for a single object.

<u>syntax</u> int InitXforObjects(SYSTEM **sys*, int *obj*, setA *P*, setA *Q*, int *matrix_type*);

<u>remarks</u> The same algorithm as InitX. Considers only elements of MATRIX D belonging to a single objects.

<u>see</u> InitX

<u>domain</u> *matrix_type*: MATA, MATD

InitXforObjectFromClass

<u>function</u> Generates MATRIX X for single class element.

<u>syntax</u> InitXforObjectFromClass(SYSTEM **sys*, int *obj*, setA *P*, setA *Q*, setO *aclass*, int *matrix_type*);

<u>remarks</u> The same algorithm as InitX. Considers only elements of MATRIX D indexed by a given *object* and any object from outside of *class*.

<u>see</u> InitX

<u>domain</u> *matrix_type*: MATA, MATD

InitXforObjects

<u>function</u> Generates MATRIX X for a set of objects.

<u>syntax</u> int InitXforObjects(SYSTEM **sys*, setA *objects*, setA *P*, setA *Q*, int *matrix_type*);

<u>remarks</u> The same algorithm as InitX. Considers only elements of MATRIX D belonging to any object from a set.

<u>see</u> InitX

domain *matrix_type*: MATA, MATD

InSet

function Tests if one set contains another.

syntax int InSetA(setA *big*, setA *small*);
int InSetO(setO *big*, setO *small*);

remarks Returns 1 if *big* contains *small*, otherwise returns 0. All parameters should be initialized in a range of the currently active information system.

InterSet

function Tests if nonempty product.

syntax int InterSetA(setA *s1*, setA *s2*);
int InterSetO(setO *s1*, setO *s2*);

remarks Returns 1 if product of *s1* and *s2 is* nonempty, otherwise returns 0. All Parameters should be initialized in a range of the currently active information system.

IsCover

function Tests if a set of attributes preserves original classification.

syntax int IsRed(setA *cov*, int *matrix_type*);

remarks Returns 1 if *cov* preserves the original classification of the active information system (contains one of reducts). Otherwise returns 0. X version has the same effect only if MATRIX X was generated with parameters P and Q containing all attributes. Otherwise it tests if *cov* preserves Q-classification of P. If error, returns negative error code.

domain *matrix_type*: MATA, MATD, MATX.

IsCoverRel

function Tests if a set of attributes preserves specified classification.

syntax int IsCoverRel(setA *cov*, setA *P*, setA *Q*, int *matrix_type*);

remarks Returns 1 if *cov* preserve *Q*-classification of objects discernible on *P* (contains *Q*-relative reduct of the active information system with attributes restricted to *P*). This reflects the situation when *P* represents condition attributes and *Q* decision attributes. *Q* does not have to be a complement of *P*. If error, returns negative error code. For X version see IsCover.

domain *matrix_type*: MATA, MATD.

IsEmpty

function	Tests if a set is empty.
syntax	int IsEmptySetA(setA *set*); Int IsEmptySetO(setO *set*);
remarks	Returns 1 if *set* is empty, otherwise returns 0.

IsOrtho

function	Tests if a set of attributes is orthogonal.
syntax	int IsOrtho(setA *ortho*, setA *over*, int *matrix_type*);
remarks	Returns 1 if all elements of *ortho* are indispensable. Otherwise returns 0. Fills *over* with all dispensable attributes. X version has the same effect only if MATRIX X was generated with parameters P and Q containing all attributes. Otherwise it tests if *orhto* is a Q-orthogonal in P. If error, returns negative error code.
domain	*matrix_type*: MATA, MATD, MATX.

IsOrthoRel

function	Tests if a set of attributes is relatively orthogonal.
syntax	int IsOrthoRel(setA *ortho*, setA *over*, setA *P*, setA *Q*, int *matrix_type*);
remarks	Returns 1 if all elements of *ortho* are *Q*-indispensable in P. Otherwise returns 0. Fills *over* with discernible attributes. This reflects the situation when *P* represents condition attributes and *Q* decision attributes. *Q* does not have to be a complement of *P*. All attributes should be initialized in the range of the same system. If error, returns negative error code. For X version see IsOrtho.
domain	*matrix_type*: MATA, MATD.

IsRed

function	Tests if a set is a reduct.
syntax	int IsRed(setA *red*,int *matrix_type*);
remarks	Returns 1 if *red* is the reduct of the active information system. Otherwise returns 0. X version has the same effect only if MATRIX X was generated with parameters P and Q containing all attributes. Otherwise it tests if *red* is a Q-relative reduct of P. If error, returns negative error code.
def.	see Red
example	InitX(...,P,Q,...); ...

a=IsRedX(red); /* a==IsRedRelD(red,P,Q);

domain *matrix_type*: (MATA,MATD,MATX)

IsRedRel

function Tests if a set is a relative reduct.

syntax int IsRedRel(setA *red*, setA *P*,setA *Q*, int *matrix_type*);

remarks Returns 1 if *red* is a *Q*-relative reduct of the active information system with attributes restricted to *P*. This reflects the situation when *P* represents condition attributes and *Q* decision attributes. *Q* does not have to be a complement of *P*. All attributes should be initialized in the range of the same system. If error, returns negative error code. For X version see IsRed.

def. see RedRel

domain *matrix_type*: MATA, MATD.

LowAppr

function Finds a lower approximation.

syntax int LowAppr(setO *appr*, setO *X*, setA *P*, int *matrix_type*);

remarks Puts to *appr* the *P*-lower approximation of *X* in the active information system. All sets should be initialized in the range of this system.

def. $\underline{P}(X) = \bigcup\{Y \in U/IND(P) : Y \subseteq X\}$

domain *matrix_type*: MATA, MATD.

MatExist

function Returns pointer to specified matrix if exists.

syntax void *MatExist(SYSTEM **sys*, int *matrix_type*);

remarks Otherwise returns NULL.

domain *matrix_type*: MATA, MATD, MATX

MatMemSize

function Returns size of memory used by matrix.

syntax unsigned int MatMemSize(SYSTEM **sys*, int *matrix_type*);

remarks MATD and MATX sizes are computed. MATX must exist.

domain *matrix_type*: MATA, MATD, MATX.

NEXT_OF_MAT (macro)

function One of macros enabling stream access to arrays of sets of attributes.

see START_OF_D, START_OF_X, START_OF_MAT

NotSet

function Finds a complement of a set.

syntax void NotSetA(setA *not*, setA *set*);

void NotSetO(setO *not*, setO *set*);

remarks Complement of *set* is put to *not*. Both parameters should be initialized in the range of the currently active system information system.

ObjectsForRule

function Finds a set of objects covered by a rule.

syntax int ObjectsForRule(setO *set*, value_type **rule*);

remarks All objects matching *rule* will be placed to *set*.

ObjectsNum

function Returns number of objects in a system.

syntax int ObjectsNum(SYSTEM **sys*);

remarks The number of objects is an information system parameter. It can be set by SetParameters() or FileToSys() routines.

OrSet

function Finds an union of sets.

syntax void OrSetA(setA *or*, setA *s1*, setA *s2*);

void OrSetO(setO *or*, setO *s1*, setO *s2*);

remarks Puts an union of *s1* and *s2* to *or*. All parameters should be initialized in a range of the currently active information system.

Pos

function Finds a positive region of classification.

syntax int Pos(setO *pos*, setA *P*, setA *Q*, int *matrix_type*);

remarks Puts to *pos a P*-positive region of *classification U\IND(Q)* in the active information system. All sets should be initialized in the range of this system. If error detected, returns negative error code.

def. $POS_P(Q) = \bigcup_{X \in U/IND(Q)} \underline{P}(X)$

domain *matrix_type*: MATA, MATD.

PrintSet

function Writes a formatted set to the standard output.

syntax void PrintSetA(setA *set*);

void PrintSetO(setO *set*);

remarks For simple interaction, enables visualization of sets. Prints elements in brackets.

PutToA

function Puts a single value to MATRIX A.

syntax void PutToA(SYSTEM *sys*, int *obj*, int *attr*, value_type *value*);

remarks If MATRIX A was connected to *sys* empty or from any other reason application wants to write to it, this routine will set *value* in the column *attr* and the row *obj*.

Red

function Finds all reducts.

syntax int Red(setA **red*, int *matrix_type*);

remarks Finds all reducts of the active information system and places them in array. It allocates memory for this array of sets of attributes and assigns *red* to its first element. *red* should be given uninitialized. For the simple access to this array see macro START_OF_MAT. You can free memory used by this array using standard free(*red*) routine. Returns number of reducts. If error, returns negative error code. X version gives the same result only if MATRIX X was generated with parameters P and Q containing all attributes. Otherwise X version finds Q-relative reducts of the information system with attributes restricted to P.

def. *RED= {a∈R: a is indispensable in RED}*
where *R= {all attributes}, IND(RED)=IND(R)*

domain *matrix_type*: MATA, MATD, MAX.

example
```
InitX(sys1,P,Q,MATD);
...
a=RedX(&red);  /* a=RedRelD(&reducts,P,Q) */
for(START_OF_MAT(red,a);END_OF_MAT;NEXT_OF_MAT)
  PrintSetA(ELEM_OF_MAT);
```

RedFew

function Finds all reducts shorter than the first one found.

syntax int RedLess(setA **red*, int *n*, int *matrix_type*);

remarks Finds all reducts that contain less attributes than the first reduct found by heuristic search. For the definition see Red.

domain *matrix_type*: MATA, MATD, MATX.

RedFirst

function	Finds first n quasi-reducts.
syntax	int RedFirst(setA **red*, int *n*, int *matrix_type*);
remarks	The routine works similarly to Red and reader should refer to its description. The difference is that the space of a search for reducts is limited to *n*. It can be implied by memory or time limits. But the construction of reduct finding algorithm causes, that received sets can be only quasi-reduct, still with the full ability to classify objects, but possibly with some attributes being dispensable. If *n* is greater than the returned number of reducts the routine works exactly like Red.
domain	*matrix_type*: MATA, MATD, MATX.

RedLess

function	Finds short reducts.
syntax	int RedLess(setA **red*, int *n*, int *matrix_type*);
remarks	Finds all reducts that contain less then *n* attributes. For the definition see Red.
domain	*matrix_type*: MATA, MATD, MATX.

RedOptim

function	Provides the optimized heuristic search for a single reduct.
syntax	int RedOptim(setA *red*, int *matrix_type*);
remarks	The reduct is stored in *red*. Dependency coefficient is used to select next potential attribute.
domain	*matrix_type*: MATA, MATD, MATX.

RedRel

function	Finds all relative reducts.
syntax	int RedRel(setA **red*, setA *P*, setA *Q*, int *matrix_type*);
remarks	Finds all *Q*-relative reducts of the active information system with attributes reduced to *P*. This reflects the situation when *P* determines condition attributes and *Q* decision attributes. *Q* does not have to be a complement of *P*. Routine places reducts in array of sets of attributes. It allocates memory for this array and assigns *red* to its first element. *red* should be given uninitialized. All other sets should be initialized in the range of the currently active system. For the simple access to this array see macro START_OF_TAB. You can free memory used by this array using standard free(*red*) routine. Returns number of reducts.

If error, returns negative error code. For X version and an example see Red.

def. *RED*$_Q$*(P)= {a∈P: a is indispensable in RED}*
where *IND(RED)=IND(P)*

domain *matrix_type*: MATA, MATD.

RedRelFew

function Finds all relative reducts shorter than the first one found.

syntax int RedRelLess(setA **red*, setA *P*, setA *Q*, int *matrix_type*);

remarks Finds all *Q*-relative reducts of the active information system with attributes restricted to *P* that have less elements than the first reduct found by heuristic search. For definition see RedRel.

domain *matrix_type*: MATA, MATD.

RedRelFirst

function Finds first n relative quasi-reducts.

syntax int RedRelFirst(setA **red*, int *n*, setA *P*, setA *Q*, int *matrix_type*);

remarks The routine works similarly to RedRel and reader should refer to its description. The difference is that the space of a search for reducts is limited to *n*. It can be implied by memory or time limits. But the construction of reduct finding algorithm causes, that received sets can be only quasi-reduct, still with the full ability to classify objects, but possibly with some attributes being dispensable. If *n* is grater than the returned number of reducts the routine works exactly like RedRel.

domain *matrix_type*: MATA, MATD.

RedRelLess

function Finds all short relative reducts.

syntax int RedRelLess(setA **red*, int *n*, setA *P*, setA *Q*, int *matrix_type*);

remarks Finds all *Q*-relative reducts of the active information system with attributes restricted to *P* that have less than *n* elements. For a definition see RedRel.

domain *matrix_type*: MATA, MATD.

RedRelOptim

function Provides the optimized heuristic search for a single relative reduct.

syntax int RedRelOptim(setA *red*, setA *P*, setA *Q*, int *matrix_type*);

remarks	The Q-relative reduct of P is stored to *red*. The dependency coefficient is used to select next potential attribute.
domain	*matrix_type*: MATA, MATD.

RedRelSetA

function	Finds all relative quasi-reducts containing a given set.
syntax	int RedRelSetA(setA **red*, setA *base*, setA *P*, setA *Q*, int *matrix_type*);
remarks	The routine works similarly to RedRel and reader should refer to its description. The difference is that it searches for Q-relative quasi-reducts of P containing attributes of *base*. They will have full ability to classify objects, but possibly some attributes from *base* will be Q-dispensable in them. If *base* is a subset of a $\mathrm{CORE}_Q(P)$ than the routine works exactly like RedRel.
domain	*matrix_type*: MATA, MATD.

RedRelSingle

function	Provides the heuristic search for a single relative reduct.
syntax	int RedRelSingle(setA *red*, setA *P*, setA *Q*, int *matrix_type*);
remarks	The Q-relative reduct of P is stored to *red*.
domain	*matrix_type*: MATA, MATD.

RedSetA

function	Finds all quasi-reducts containing a given set.
syntax	int RedSetA(setA **red*, setA *base*, int *matrix_type*);
remarks	The routine works similarly to Red and reader should refer to its description. The difference is that it searches for quasi-reducts containing *base*. This quasi-reducts will have full ability to classify objects, but possibly some attributes from *base* will be dispensable. If *base* is a subset of a CORE than the routine works exactly like Red.
domain	*matrix_type*: MATA, MATD, MATX.

RedSingle

function	Provides the heuristic search for a single reduct.
syntax	int RedSingle(setA *red*, int *matrix_type*);
remarks	The reduct is stored in *red*.
domain	*matrix_type*: MATA, MATD, MATX.

RuleCopy

function	Copies a single rule.
syntax	void RuleCopy(value_type **dest*, value_type **source*);
remarks	Copies one rule from *source* to *dest*.

RuleEQ

function	Compares two rule.
syntax	void RuleEQ(value_type **first*, value_type **second*);
remarks	Return 1 if *first* is equal to *second*. Otherwise returns 0.

Rules

function	Finds a collection of rules.
syntax	int Rules(value_type ***rules*, setA *P*, setA *Q*, int *matrix_type*);
remarks	For each object from the active information system finds all the shortest *Q*-relative reducts of values on *P*. Selects the set of rules to cover all objects. Allocates memory and returns number of generated rules.
domain	*matrix_type*: MATA, MATD.

RulesForClass

function	Finds a collection of rules.
syntax	int RulesForClass(value_type ***rules*, setO *class*, setA *P*, setA *Q*, int *matrix_type*);
remarks	For each object from the *class* finds all the shortest *Q*-relative reducts of values on *P*. Selects the set of rules to cover the *class*. Allocates memory and returns number of generated rules.
domain	*matrix_type*: MATA, MATD.

SelectAllShort

function	Copies the shortest reducts.
syntax	int SelectAllShort(setA **newred*, setA *reds*, int *num*);
remarks	Creates a collection of all the shortest reducts selected from *newred* collection of existing *num* reducts. Allocates memory and returns number of selected reducts.

SelectOneShort

function	Selects the shortest reduct.
syntax	int SelectOneShort(setA *reds*, int *num*);

remarks	Returns index of the first shortest reduct from the collection *reds* of *num* reducts.

SelectRules

function	Reduce number of rules.
syntax	int SelectRules(value_type ***rules*, int **n*, setO *objects*, setA *P*, int *option*);
remarks	Reduce number of rules to cover only given *objects*. Reallocate memory for *rules* and decrease *n*.
option	BESTOPT - optimal selection FASTOPT - optimized for speed.

SetName

function	Sets an information system name.
syntax	void SetName(SYSTEM **sys*, char **name*);
remarks	Copies string *name* as a system name to *sys* (maximum length 50).

SetParameters

function	Sets information system parameters.
syntax	void SetParameters(SYSTEM **sys*, int *objects_num*, int *attributes num*);
remarks	Sets number of attributes and number of objects in *sys*. This parameters will determine the sizes of data matrices and sets representation.

SignifCoef

function	Computes a significance of an attribute.
syntax	float SignifCoef(int *attr*, setA *P*, int *matrix_type*);
remarks	Returns the significance of *attr* in the set of attributes *P* in the active information system. *P* should be initialized in the range of this system.
def.	$\mu_P(attr) = card(POS_P(R) - POS_{P-\{attr\}}(R))\ /$ $card(all\ objects)$ where $R = \{all\ attributes\}$
domain	*matrix_type*: MATA, MATD.

SignifRelCoef

function	Computes relative significance of an attribute.
syntax	float SignifRelCoef(int *attr*, setA *P*, setA *Q*, int *matrix_type*);
remarks	Returns the significance of *attr* in the set of attributes *P* relatively to *Q* in the active information system. Sets should be initialized in the range of the same system.
def.	$\mu_{P,Q}(attr) = card(POS_P(Q)) - POS_{P-\{attr\}}(Q))\ /$

card({all objects})

domain *matrix_type*: MATA, MATD

SingCompA

function Compares two objects on a single attribute.

syntax int SingCompA(int *obj1*, int *obj2*, int *attr*);

remarks Returns 1 if *obj1* and *obj2* have the same value of attribute in MATRIX A of the active information system. Otherwise returns 0.

SingCompD

function Test for discernibility of two objects on a single attribute.

syntax int SingCompD(int *obj1*, int *obj2*, int *attr*);

remarks Returns 1 if *obj1* and *obj2* are indiscernible on attribute *attr* in MATRIX D of the active information system. Otherwise returns 0.

SizeSet

function Returns the number of cluster_type elements used for set representation.

syntax int SizeSetA(void);
int SizeSetO(void);

remarks Result determined by the active information system parameters.

START_OF_D (macro)

function Enables stream access to elements (sets of attributes) of MATRIX D.

syntax START_OF_D;

remarks This macro was designed to form, together with other macros, a for(...) loop. ELEM_OF_TAB , type of *setA* indicates the following element of MATRIX D. See an example. Such loops can not be nested. Memory size for sets and MATRIX D determined by the active information system.

example
```
for(START_OF_D;END_OF_MAT;NEXT_OF_MAT)
  PrintSetA(ELEM_OF_MAT);
/* all elements of MATRIX D will be printed */
/* in order of existence in memory */
```

START_OF_MAT (macro)

function Enable stream access to an array of sets of attributes.

syntax START_OF_MAT(setA *first*,int *num*);

remarks This macro was designed to form, together with other macros, a for(...) loop. It is the easiest access to arrays of sets of attributes that were allocated and filled by reducts queries. See an example. Such loops can not be nested. ELEM_OF_MAT ,type of setA, will indicate *num* following sets of attributes starting with *first*.

example
```
setA reducts;
int i;
....
i=RedA(&reducts);
for(START_OF_MAT(reducts,num);END_OF_MAT;NEXT_OF_MAT
)
  PrintSetA(ELEM_OF_MAT);
/* all reducts will be printed */
/* in order of existence in memory */
```

START_OF_X (macro)

function Enables stream access to elements (sets of attributes) of MATRIX X.

syntax START_OF_X;

remarks This macro was designed to form, together with other macros, a for(...) loop. ELEM_OF_MAT, type of *setA*, indicates the following element of MATRIX X. See an example. Such loops can not be nested. Memory size of sets and MATRIX X from active information system.

example
```
for(START_OF_X;END_OF_MAT;NEXT_OF_MAT)
  PrintSetA(ELEM_OF_MAT);
/* all elements of MATRIX X will be printed */
/* in order of existence in memory */
```

StrengthOfRule

function Returns a number of objects covered by a rule.

syntax int StrengthOfRule(value_type **rule*);

remarks Objects must match on all unreduced attributes, both conditions and decisions.

StrengthOfRules

function Creates a table of rules strengths.

syntax int *StrengthOfRules(value_type **rules*, int *N*);

remarks — Strength is the number objects matching on all unreduced attributes, both conditions and decisions. Allocates memory (size = *N**sizeof(int)).

SysName

function — Returns a name of a system.
syntax — char *SysName(SYSTEM **sys*);
remarks — Returns system name string from *sys* (maximum length 50). It returns the pointer to a descriptor field, so it should be read only.

SysToFile

function — Exports an information system to a file of special format.
syntax — int SysToFile(SYSTEM **sys*, char **filename*);
remarks — For the precise description of the file format see chapter Data Structures. The file contains information system parameters, name, additional description and MATRIX A. If any error, returns negative error code.

TabToSet

function — Puts elements of a table to a set.
syntax — void TabToSetA(setA *set*, int *num*, int *tab*[]);
void TabToSetO(setO *set*, int *num*, int *tab*[]);
remarks — Puts *num* elements of *tab* to *set*. Old contents of *set* will be erased.

UppAppr

function — Finds an upper approximation.
syntax — int UppAppr(setO *appr*, setO *X*, setA *P*, int *matrix_type*);
remarks — Puts to *appr* the *P*-upper approximation of *X* in the active information system. All sets should be initialized in the range of this system.
def. — $\overline{P}(X) = \bigcup\{ Y \in U/IND(P) : Y \cap X \neq \emptyset \}$
domain — *matrix_type*: MATA, MATD.

UseSys

function — Activates a system.
syntax — void UseSys(SYSTEM **sys*);
remarks — From this moment on, all routines will work on this indicated system data and parameters.

VeryFastRules

function	Finds very quickly a collection of rules.
syntax	int VeryFastRules(value_type ***rules*, setA *P*, setA *Q*, int *matrix_type*);
remarks	For each object from the active information system finds a single *Q*-relative reduct of values on *P* (heuristic search). Removes repetitions. Allocates memory and returns number of generated rules.
domain	*matrix_type*: MATA, MATD.

Xsize

function	Returns number of elements (setA) used by MATRIX X.
syntax	unsigned int Xsize(SYSTEM **sys*);
remarks	MATRIX X should be initialized (generated).

APPENDIX A : SOURCE FILES

The source of the library routines is divided into several files. File segregation corresponds to task segregation. Each **.c* should be compiled separately. Received object files form the library.

Source files of the library modules:

rsystem.c	- system control
rset.c	- set handling
raccess.c	- data access
rbasic.c	- simple queries
rcore.c	- core queries
reduct1.c	- reduct queries part 1
reduct2.c	- reduct queries part 2
rule1.c	- rule generation part 1
rule2.c	- rules generation part 2
rclass.c	- classification

All routines and global variables defined in **.c* file are declared in corresponding **.h* file.

Declaration files:

rsystem.h
rset.h
raccess.h
rbasic.h
rcore.h
reduct1.h
reduct2.h
rule1.h
rule2.h
rclass.h

Global types and constants are defined in the file *rough.h*. This file includes, by preprocessor commands, all declaration files. Thus, it contains all the definitions and declarations necessary for the whole library. It is the only and obligatory header file to be included in every application source.

Header file:

rough.h

File *rerror.h* contains error messages and is not used by any library routine. For description see chapter ERROR HANDLING.

APPENDIX B: INSTALLING IN MS-DOS

The library is available in ANSI source code (see APPENDIX A). It can be compiled and installed in MS-DOS or any other system without any changes. Since it does not supply any INPUT/OUTPUT functions there are no special requirements for monitor type. The library routines use only ANSI standard functions so any compiler is probably going to be adequate. However, for successful work in the standard PC architecture the library has to be slightly altered.

The problem may be memory allocation. Some of the RSL data structures and especially MATRIX D require large continuous memory blocks. The library use only standard *malloc()* and *free()* functions. All memory sizes are stored as *int* values. When the size of MATRIX D or ADDITIONAL DESCRIPTION does not exceed the domain of *int* the library works properly. When it may become bigger, some routines and data structure have to be modified. When it exceeds the whole memory available for the application only matrices A and X can be used. Disk swapping would require interference with the work of the whole library.

memory requirement for MATRIX D :

$$N * (N-1)/2 * (1+ (M-1) \bmod 8)$$

where:

N = number of attributes in system

M = number of objects in system

APPENDIX C: EXAMPLE OF DATA FILE

```
NAME: persons
ATTRIBUTES: 4
OBJECTS: 8

0 0 0 0
1 0 1 ?
1 1 0 0
0 2 0 1
1 ? 0 1
1 0 0 0
1 2 ? 1
0 0 1 1

CODING TABLE:
0 - Height
     0 - short
     1 - tall
1 - Hair
     0 - blond
     1 - red
     2 - dark
2 - Eyes
     0 - blue
     1 - brown
3 - Attractiveness
     0 - plus
     1 - minus
```

BIBLIOGRAPHY

Pawlak'82 Pawlak Z. *Rough Sets*, International Journal of Information and Computer Science Vol. 11, No. 5, 1982, pp. 344-356.

Pawlak'91 Pawlak Z. *Rough Sets, Theoretical Aspects of Reasoning about Data*, Kluwer Academic Publishers, 1991.

Skowron'92 Skowron A., Rauszer C. *The Discernibility Matrices and Functions in Information Systems*, In: Słowiński R. (ed.), Decision Support by Experience - Applications of the Rough Sets Theory, Kluwer 1992, pp. 331-362.

Słowiński'92 Słowiński R., Stefanowski J. *'ROUGHDAS' and 'ROUGHCLASS' Software Implementation of the Rough Sets Approach*, In: R. Słowiński (ed.) Intelligent Decision Support. Handbook of Applications and Advances of the Rough Sets Theory, Kluwer Academic Publishers, 1992.